# Inter-iteration Scalar Replacement Using Array SSA Form

Rishi Surendran[1], Rajkishore Barik[2], Jisheng Zhao[1], and Vivek Sarkar[1]

[1] Rice University, Houston, TX
{rishi,jisheng.zhao,vsarkar}@rice.edu
[2] Intel Labs, Santa Clara, CA
rajkishore.barik@intel.com

**Abstract.** In this paper, we introduce novel simple and efficient analysis algorithms for scalar replacement and dead store elimination that are built on Array SSA form, a uniform representation for capturing control and data flow properties at the level of array or pointer accesses. We present extensions to the original Array SSA form representation to capture loop-carried data flow information for arrays and pointers. A core contribution of our algorithm is a subscript analysis that propagates array indices across loop iterations. Compared to past work, this algorithm can handle control flow within and across loop iterations and degrade gracefully in the presence of unanalyzable subscripts. We also introduce code transformations that can use the output of our analysis algorithms to perform the necessary scalar replacement transformations (including the insertion of loop prologues and epilogues for loop-carried reuse). Our experimental results show performance improvements of up to 2.29× relative to code generated by LLVM at -O3 level. These results promise to make our algorithms a desirable starting point for scalar replacement implementations in modern SSA-based compiler infrastructures such as LLVM.

**Keywords:** Static Single Assignment (SSA) form, Array SSA form, Scalar Replacement, Load Elimination, Store Elimination.

## 1 Introduction

Scalar replacement is a widely used compiler optimization that promotes memory accesses, such as a read of an array element or a load of a pointer location, to reads and writes of compiler-generated temporaries. Current and future trends in computer architecture provide an increased motivation for scalar replacement because compiler-generated temporaries can be allocated in faster and more energy-efficient storage structures such as registers, local memories and scratchpads. However, scalar replacement algorithms in past work [6,9,7,3,14,4,2,21,5] were built on non-SSA based program representations, and tend to be complex to understand and implement, expensive in compile-time resources, and limited in effectiveness in the absence of precise data dependences. Though the benefits of SSA-based analysis are well known and manifest in modern compiler

A. Cohen (Ed.): CC 2014, LNCS 8409, pp. 40–60, 2014.

infrastructures such as LLVM [13], it is challenging to use SSA form for scalar replacement analysis since SSA form typically focuses on scalar variables and scalar replacement focuses on array and pointer accesses.

In this paper, we introduce novel simple and efficient analysis algorithms for scalar replacement and dead store elimination that are built on Array SSA form [12], an extension to scalar SSA form that captures control and data flow properties at the level of array or pointer accesses. We present extensions to the original Array SSA form representation to capture loop-carried data flow information for arrays and pointers. A core contribution of our algorithm is a subscript analysis that propagates array indices across loop iterations. Compared to past work, this algorithm can handle control flow within and across loop iterations and degrades gracefully in the presence of unanalyzable subscript. We also introduce code transformations that can use the output of our analysis algorithms to perform the necessary scalar replacement transformations (including the insertion of loop prologs and epilogues for loop-carried reuse). These results promise to make our algorithms a desirable starting point for scalar replacement implementations in modern SSA-based compiler infrastructures.

The main contributions of this paper are:

- Extensions to Array SSA form to capture inter-iteration data flow information of arrays and pointers
- A framework for inter-iteration subscript analysis for both forward and backward data flow problems
- An algorithm for inter-iteration redundant load elimination analysis using our extended Array SSA form, with accompanying transformations for scalar replacement, loop prologs and loop epilogues.
- An algorithm for dead store elimination using our extended Array SSA form, with accompanying transformations.

The rest of the paper is organized as follows. Section 2 discusses background and motivation for this work. Section 3 contains an overview of scalar replacement algorithms. Section 4 introduces Array SSA form and extensions for inter-iteration data flow analysis. Section 5 presents available subscript analysis, an inter-iteration data flow analysis. Section 6 describes the code transformation algorithm for redundant load elimination, and Section 7 describes the analysis and transformations for dead store elimination. Section 8 briefly summarizes how our algorithm can be applied to objects and while loops. Section 9 contains details on the LLVM implementation and experimental results. Finally, Section 10 presents related work and Section 11 contains our conclusions.

## 2 Background

In this section we summarize selected past work on scalar replacement which falls into two categories. 1) inter-iteration scalar replacement using non-SSA representations and 2) intra-iteration scalar replacement using Array SSA form, to provide the background for our algorithms. A more extensive comparison with related work is presented later in Section 10.

| (a) Original Loop | (b) After Scalar Replacement |
|---|---|
| 1: **for** $i = 1$ to $n$ **do**<br>2: $B[i] = 0.3333 * (A[i-1] + A[i] + A[i+1])$<br>3: **end for** | 1: $t_0 = A[0]$<br>2: $t_1 = A[1]$<br>3: **for** $i = 1$ to $n$ **do**<br>4: $t_2 = A[i+1]$<br>5: $B[i] = 0.3333 * (t_0 + t_1 + t_2)$<br>6: $t_0 = t_1$<br>7: $t_1 = t_2$<br>8: **end for** |

**Fig. 1.** Scalar replacement on a 1-D Jacobi stencil computation [1]

### 2.1 Inter-iteration Scalar Replacement

Figure 1(a) shows the innermost loop of a 1-D Jacobi stencil computation [1]. The number of memory accesses per iteration in the loop is four, which includes three loads and a store. The read references involving array $A$ present a reuse opportunity in that the data read by $A[i+1]$ is also read by $A[i]$ in the next iteration of the loop. The same element is also read in the following iteration by $A[i-1]$. The reference $A[i+1]$ is referred to as the *generator* [7] for the redundant loads, $A[i]$ and $A[i-1]$. The number of memory accesses inside the loop could thus be reduced to one, if the data read by $A[i+1]$ is stored in a scalar temporary which could be allocated to faster memory. Assuming $n > 0$, the loop after scalar replacement transformation is shown in 1(b). Non-SSA algorithms for inter-iteration scalar replacement have been presented in past work including [6,7,9]. Of these, the work by Carr and Kennedy [7] is described below, since it is the most general among past algorithms for inter-iteration scalar replacement.

### 2.2 Carr-Kennedy Algorithm

The different steps in the Carr-Kennedy algorithm [7] are 1) Dependence graph construction, 2) Control flow analysis, 3) Availability analysis, 4) Reachability analysis, 5) Potential generator selection, 6) Anticipability analysis, 7) Dependence graph marking, 8) Name partitioning, 9) Register pressure moderation, 10) Reference replacement, 11) Statement insertion analysis, 12) Register copying, 13) Code motion, and 14) Initialization of temporary variables.

The algorithm is complex, requires perfect dependence information to be applicable and operates only on loop bodies without any backward conditional flow. Further, the algorithm performs its profitability analysis on *name partitions*, where a name partition consists of references that share values. If a name partition is selected for scalar replacement, all the memory references in that name partition will get scalar replaced, otherwise none of the accesses in the name partition are scalar replaced.

### 2.3 Array SSA Analysis

Array SSA is a program representation which captures precise element-level data-flow information for array variables. Every use and definition in the extended Array SSA form has a unique name. There are 3 different types of $\phi$ functions presented in [10].

1. A *control* $\phi$ (denoted simply as $\phi$) corresponds to the $\phi$ function from scalar SSA. A $\phi$ function is added for a variable at a join point if multiple definitions of the variable reach that point.
2. A *definition* $\phi$ ($d\phi$) [12] is used to deal with *partially killing* definitions. A $d\phi$ function of the form $A_k = d\phi(A_i, A_j)$ is inserted immediately after each definition of the array variable, $A_i$, that does not completely kill the array value. $A_j$ is the augmenting definition of $A$ which reaches the point just prior to the definition of $A_i$. A $d\phi$ function merges the value of the element modified with the values that are available prior to the definition.
3. A *use* $\phi$ ($u\phi$) [10] function creates a new name whenever a statement reads an array element. The purpose of the $u\phi$ function is to link together uses of the same array in control-flow order. This is used to capture the read-after-read reuse (aka input dependence). A $u\phi$ function of the form $A_k = u\phi(A_i, A_j)$ is inserted immediately after the use of an array element, $A_i$. $A_j$ is the augmenting definition of $A$ which reaches the point just prior to the use of $A_i$.

[10] presented a unified approach for the analysis and optimization of object field and array element accesses in strongly typed languages using Array SSA form. But the approach had a major limitation in that it does not capture reuse across loop iterations. For instance, their approach cannot eliminate the redundant memory accesses in the loop in Figure 1. In Section 4, we introduce extensions to Array SSA form for inter-iteration analysis.

### 2.4 Definitely-Same and Definitely-Different Analyses

In order to reason about aliasing among array accesses, [10] describes two relations: $\mathcal{DS}$ represents the *Definitely-Same* binary relationship and $\mathcal{DD}$ represents the *Definitely-Different* binary relationship. $\mathcal{DS}(a, b) = true$ if and only if $a$ and $b$ are guaranteed to have the same value at all program points that are dominated by the definition of $a$ and dominated by the definition of $b$. Similarly, $\mathcal{DD}(a, b) = true$ if and only if $a$ and $b$ are guaranteed to have different values at all program points that are dominated by the definition of $a$ and dominated by the definition of $b$. The Definitely-same ($\mathcal{DS}$) and Definitely-different ($\mathcal{DD}$) relation between two array subscripts can be computed using different methods and is orthogonal to the analysis and transformation described in this paper.

## 3 Scalar Replacement Overview

In this section, we present an overview of the basic steps of our scalar replacement algorithms: redundant load elimination and dead store elimination. To simplify

the description of the algorithms, we consider only a single loop. We also assume that the induction variable of the loop has been normalized to an increment of one. Extensions to multiple nested loops can be performed in hierarchical fashion, starting with the innermost loop and analyzing a single loop at a time. When analyzing an outer loop, the array references in the enclosed nested loops are summarized with subscript information [16].

The scalar replacement algorithms include three main steps:

1. *Extended Array SSA Construction*:
   In the first step, the extended Array SSA form of the original program is constructed. All array references are renamed and $\phi$ functions are introduced as described in Section 4. Note that the extended Array SSA form of the program is used only for the analysis (presented in step 2). The transformations (presented in step 3) are applied on the original program.
2. *Subscript analysis*:
   Scalar replacement of array references is based on two subscript analyses: (a) *available subscript analysis* identifies the set of redundant loads in the given loop, which is used for redundant load elimination (described in Section 6); (b) *dead subscript analysis* identifies the set of dead stores in the given loop, which is used in dead store elimination (described in Section 7). These analyses are performed on extended Array SSA form and have an associated tuning parameter: the maximum number of iterations for which the analysis needs to run.
3. *Transformation*:
   In this step, the original program is transformed using the information produced by the analyses described in step 2. For redundant load elimination, this involves replacing the read of array elements with read of scalar temporaries, generating copy statements for scalar temporaries and generating statements to initialize the temporaries. The transformation is presented in Section 6. Dead store elimination involves removing redundant stores and generating epilogue code as presented in Section 7.

## 4 Extended Array SSA Form

In order to model inter-iteration reuse, the lattice operations of the $\phi$ function in the loop header needs to be handled differently from the rest of the control $\phi$ functions. They need to capture what array elements are available from prior iterations. We introduce a *header* $\phi$ ($h\phi$) node in the loop header. We assume that every loop has one incoming edge from outside and thus, one of the

```
1: for i = 1 to n do
2:     if A[B[i]] > 0 then
3:         A[i+1] = A[i-1] + B[i-1]
4:     end if
5:     A[i] = A[i] + B[i] + B[i+1]
6: end for
```

**Fig. 2.** Loop with redundant loads and stores

arguments to the $h\phi$ denotes the SSA name from outside the loop. For each back edge from within the loop, there is a corresponding SSA operand added to the $h\phi$ function. Figure 2 shows a loop from [11, p. 387] extended with control flow. The three address code of the same program is given in 3(a) and the extended Array SSA form is given in 3(b). $A_1 = h\phi(A_0, A_{12})$ and $B_1 = h\phi(B_0, B_{10})$ are the two $h\phi$ nodes introduced in the loop header. $A_0$ and $B_0$ contain the definitions of array $A$ which reaches the loop preheader.

While constructing Array SSA form, $d\phi$ and $u\phi$ functions are introduced first into the program. The control $\phi$ and $h\phi$ functions are added in the second phase. This will ensure that the new SSA names created due to the insertion of $u\phi$ and $d\phi$ nodes are handled correctly. We introduce at most one $d\phi$ function for each array definition and at most one $u\phi$ function for each array use. Past work have shown that the worst-case size of the extended Array SSA form is proportional to the size of the scalar SSA form that would be obtained if each array access is modeled as a definition [10]. Past empirical results have shown the size of scalar SSA form to be linearly proportional to the size of the input program [8].

| **(a) Three Address Code** | **(b) Array SSA form** |
|---|---|
| 1: **for** $i = 1$ to $n$ **do** | 1: $A_0 = ...$ |
| 2: $t1 = B[i]$ | 2: $B_0 = ...$ |
| 3: $t2 = A[t1]$ | 3: **for** $i = 1$ to $n$ **do** |
| 4: **if** $t2 > 0$ **then** | 4: $A_1 = h\phi(A_0, A_{12})$ |
| 5: $t_3 = A[i-1]$ | 5: $B_1 = h\phi(B_0, B_{10})$ |
| 6: $t_4 = B[i-1]$ | 6: $t1 = B_2[i]$ |
| 7: $t_5 = t_3 + t_4$ | 7: $B_3 = u\phi(B_2, B_1)$ |
| 8: $A[i+1] = t_5$ | 8: $t2 = A_2[t1]$ |
| 9: **end if** | 9: $A_3 = u\phi(A_2, A_1)$ |
| 10: $t_6 = A[i]$ | 10: **if** $t2 > 0$ **then** |
| 11: $t_7 = B[i]$ | 11: $t_3 = A_4[i-1]$ |
| 12: $t_8 = B[i+1]$ | 12: $A_5 = u\phi(A_4, A_3)$ |
| 13: $t_9 = t_6 + t_7$ | 13: $t_4 = B_4[i-1]$ |
| 14: $t_{10} = t_9 + t_8$ | 14: $B_5 = u\phi(B_4, B_3)$ |
| 15: $A[i] = t_{10}$ | 15: $t_5 = t_3 + t_4$ |
| 16: **end for** | 16: $A_6[i+1] = t_5$ |
| | 17: $A_7 = d\phi(A_6, A_5)$ |
| | 18: **end if** |
| | 19: $A_8 = \phi(A_3, A_7)$ |
| | 20: $B_6 = \phi(B_3, B_5)$ |
| | 21: $t_6 = A_9[i]$ |
| | 22: $A_{10} = u\phi(A_9, A_8)$ |
| | 23: $t_7 = B_7[i]$ |
| | 24: $B_8 = u\phi(B_7, B_6)$ |
| | 25: $t_8 = B_9[i+1]$ |
| | 26: $B_{10} = u\phi(B_9, B_8)$ |
| | 27: $t_9 = t_6 + t_7$ |
| | 28: $t_{10} = t_9 + t_8$ |
| | 29: $A_{11}[i] = t_{10}$ |
| | 30: $A_{12} = d\phi(A_{11}, A_{10})$ |
| | 31: **end for** |

**Fig. 3.** Example Loop and extended Array SSA form

## 5 Available Subscript Analysis

In this section, we present the subscript analysis which is one of the key ingredients for inter-iteration redundant load elimination (Section 6) and dead store elimination transformation (Section 7). The subscript analysis takes as input the extended Array SSA form of the program and a parameter, $\tau$, which represents the maximum number of iterations across which inter-iteration scalar replacement will be applied on. An upper bound on $\tau$ can be obtained by computing the maximum dependence distance for the given loop, when considering all dependences in the loop. However, since smaller values of $\tau$ may sometimes be better

due to register pressure moderation reasons, our algorithm views $\tau$ as a tuning parameter. This paper focuses on the program analysis foundations of our scalar replacement approach — it can be combined with any optimization strategy for making a judicious choice for $\tau$.

Our analysis computes the set of array elements that are available at all the $\phi$, $u\phi$, $d\phi$ and $h\phi$ nodes. The lattice element for an array variable, $A$, is represented as $\mathcal{L}(A)$. The set denoted by $\mathcal{L}(A)$, represented as $\text{SET}(\mathcal{L}(A))$, is a subset of $\mathcal{U}^A_{ind} \times \mathbb{Z}_{\geq 0}$, where $\mathcal{U}^A_{ind}$ denotes the universal set of index values for $A$ and $\mathbb{Z}_{\geq 0}$ denotes the set of all non-negative integers. The lattice elements are classified as:

1. $\mathcal{L}(A_j) = \top \Rightarrow \text{SET}(\mathcal{L}(A_j)) = \mathcal{U}^A_{ind} \times \mathbb{Z}_{\geq 0}$
   This case means that all the elements of $A$ are available at $A_j$.
2. $\mathcal{L}(A_j) = \langle (i_1, d_1), (i_2, d_2) ... \rangle \Rightarrow \text{SET}(\mathcal{L}(A_j)) = \{(i_1, d_1), (i_2, d_2), ...\}$
   This means that the array element $A[i_1]$ is available at $A_j$ and is generated in the $k - d_1$th iteration, where $k$ denotes the current iteration. Similarly $A[i_2]$ is available at $A_j$ and is generated in the $k - d_2$th iteration and so on. $d_1$, $d_2$, ... is used to track the number of iterations that have passed since the corresponding array element was referenced.
3. $\mathcal{L}(A_j) = \bot \Rightarrow \text{SET}(\mathcal{L}(A_j)) = \{\}$
   This case means that, according to the current stage of analysis none of the elements in $A$ are available at $A_j$.

The lattice element computations for the SSA nodes is defined in terms of SHIFT, JOIN, INSERT and UPDATE operations. The SHIFT operation is defined as follows, where $step_1$ denotes the coefficient of the induction variable in $i_1$, $step_2$ denotes the coefficient of the induction variable in $i_2$ and so on.

$$\text{SHIFT}(\{(i_1, d_1), (i_2, d_2), \ldots\}) = \{(i_1 - step_1, d_1 + 1), (i_2 - step_2, d_2 + 1), \ldots\}$$

The definitions of JOIN, INSERT and UPDATE operations are given below.

$$\text{JOIN}(\mathcal{L}(A_p), \mathcal{L}(A_q)) = \{(i_1, d) | (i_1, d_1) \in \mathcal{L}(A_p) \text{ and } \exists\ (i'_1, d'_1) \in \mathcal{L}(A_q) \text{ and } \mathcal{DS}(i_1, i'_1) = true \text{ and } d = max(d_1, d'_1)\}$$

$$\text{INSERT}((i', d'), \mathcal{L}(A_p)) = \{(i_1, d_1) | (i_1, d_1) \in \mathcal{L}(A_p) \text{ and } \mathcal{DD}(i', i_1) = true\} \cup \{(i', d')\}$$

$$\text{UPDATE}((i', d'), \mathcal{L}(A_p)) = \{(i_1, d_1) | (i_1, d_1) \in \mathcal{L}(A_p) \text{ and } \mathcal{DS}(i', i_1) = false\} \cup \{(i', d')\}$$

Figures 4, 5, 6 and 7 describe the lattice element computations for the SSA nodes corresponding to $d\phi$, $u\phi$, $\phi$, and $h\phi$ respectively. The lattice values are initialized as follows:

$$\mathcal{L}(A_i) = \begin{cases} \{(x, 0)\} & A_i \text{ is a definition of the form } A_i[x] \\ \{(x, 0)\} & A_i \text{ is a use of the form } A_i[x] \\ \top & A_i \text{ is defined outside the loop} \\ \bot & A_i \text{ is a SSA definition inside the loop} \end{cases}$$

Figure 8 illustrates available subscript analysis on the loop in Figure 3.

We now present a brief complexity analysis of the available subscript analysis. Let $k$ be the total number of loads and stores of different array elements inside a loop. The number of $d\phi$ and $u\phi$ nodes inside the loop will be $O(k)$. Based on past empirical measurements for scalar SSA form [8], we can expect that the total number of $\phi$ nodes created will be $O(k)$. Our subscript analysis involves $\tau$ iterations in the SSA graph [8]. Therefore, in practice the complexity of the available subscript analysis is $O(\tau \times k)$, for a given loop.

| $\mathcal{L}(A_r)$ | $\mathcal{L}(A_p) = \top$ | $\mathcal{L}(A_p) = \langle (i_1, d_1), \ldots \rangle$ | $\mathcal{L}(A_p) = \bot$ |
|---|---|---|---|
| $\mathcal{L}(A_q) = \top$ | $\top$ | $\top$ | $\top$ |
| $\mathcal{L}(A_q) = \langle (i', d') \rangle$ | $\top$ | INSERT$((i', d'), \langle (i_1, d_1), \ldots \rangle)$ | $\langle (i', d') \rangle$ |
| $\mathcal{L}(A_q) = \bot$ | $\bot$ | $\bot$ | $\bot$ |

**Fig. 4.** Lattice computation for $\mathcal{L}(A_r) = \mathcal{L}_{d\phi}(\mathcal{L}(A_q), \mathcal{L}(A_p))$ where $A_r := d\phi(A_q, A_p)$ is a definition $\phi$ operation

| $\mathcal{L}(A_r)$ | $\mathcal{L}(A_p) = \top$ | $\mathcal{L}(A_p) = \langle (i_1, d_1), \ldots \rangle$ | $\mathcal{L}(A_p) = \bot$ |
|---|---|---|---|
| $\mathcal{L}(A_q) = \top$ | $\top$ | $\top$ | $\top$ |
| $\mathcal{L}(A_q) = \langle (i', d') \rangle$ | $\top$ | UPDATE$((i', d'), \langle (i_1, d_1), \ldots \rangle)$ | $\mathcal{L}(A_1)$ |
| $\mathcal{L}(A_q) = \bot$ | $\top$ | $\mathcal{L}(A_p)$ | $\bot$ |

**Fig. 5.** Lattice computation for $\mathcal{L}(A_r) = \mathcal{L}_{u\phi}(\mathcal{L}(A_q), \mathcal{L}(A_p))$ where $A_r := u\phi(A_q, A_p)$ is a use $\phi$ operation

| $\mathcal{L}(A_r) = \mathcal{L}(A_q) \sqcap \mathcal{L}(A_p)$ | $\mathcal{L}(A_p) = \top$ | $\mathcal{L}(A_p) = \langle (i_1, d_1), \ldots \rangle$ | $\mathcal{L}(A_p) = \bot$ |
|---|---|---|---|
| $\mathcal{L}(A_q) = \top$ | $\top$ | $\mathcal{L}(A_p)$ | $\bot$ |
| $\mathcal{L}(A_q) = \langle (i'_1, d'_1), \ldots \rangle$ | $\mathcal{L}(A_q)$ | JOIN$(\mathcal{L}(A_q), \mathcal{L}(A_p))$ | $\bot$ |
| $\mathcal{L}(A_q) = \bot$ | $\bot$ | $\bot$ | $\bot$ |

**Fig. 6.** Lattice computation for $\mathcal{L}(A_r) = \mathcal{L}_{\phi}(\mathcal{L}(A_q), \mathcal{L}(A_p))$, where $A_r := \phi(A_q, A_p)$ is a control $\phi$ operation

| $\mathcal{L}(A_r)$ | $\mathcal{L}(A_p) = \top$ | $\mathcal{L}(A_p) = \langle (i_1, d_1), \ldots \rangle$ | $\mathcal{L}(A_p) = \bot$ |
|---|---|---|---|
| $\mathcal{L}(A_q) = \top$ | $\top$ | $\mathcal{L}(A_p)$ | $\bot$ |
| $\mathcal{L}(A_q) = \langle (i'_1, d'_1), \ldots \rangle$ | SHIFT$(\mathcal{L}(A_q))$ | JOIN(SHIFT$(\mathcal{L}(A_q)), \mathcal{L}(A_p))$ | $\bot$ |
| $\mathcal{L}(A_q) = \bot$ | $\bot$ | $\bot$ | $\bot$ |

**Fig. 7.** Lattice computation for $\mathcal{L}(A_r) = \mathcal{L}_{h\phi}(\mathcal{L}(A_q), \mathcal{L}(A_p))$, where $A_r := h\phi(A_q, A_p)$ is a header $\phi$ operation

| | Iteration 1 | Iteration 2 |
|---|---|---|
| $\mathcal{L}(A_1)$ | $\perp$ | $\{(i-1,1)\}$ |
| $\mathcal{L}(B_1)$ | $\perp$ | $\{(i-1,1),(i,1)\}$ |
| $\mathcal{L}(B_3)$ | $\{(i,0)\}$ | $\{(i-1,1),(i,0)\}$ |
| $\mathcal{L}(A_3)$ | $\{(t,0)\}$ | $\{(i-1,1),(t,0)\}$ |
| $\mathcal{L}(A_5)$ | $\{(i-1,0),(t,0)\}$ | $\{(i-1,0),(t,0)\}$ |
| $\mathcal{L}(B_5)$ | $\{(i-1,0),(i,0)\}$ | $\{(i-1,0),(i,0)\}$ |
| $\mathcal{L}(A_7)$ | $\{(i-1,0),(i+1,0)\}$ | $\{(i-1,0),(i+1,0)\}$ |
| $\mathcal{L}(A_8)$ | $\perp$ | $\{(i-1,1)\}$ |
| $\mathcal{L}(B_6)$ | $\{(i,0)\}$ | $\{(i-1,1),(i,0)\}$ |
| $\mathcal{L}(A_{10})$ | $\{(i,0)\}$ | $\{(i-1,1),(i,0)\}$ |
| $\mathcal{L}(B_8)$ | $\{(i,0)\}$ | $\{(i-1,1),(i,0)\}$ |
| $\mathcal{L}(B_{10})$ | $\{(i,0),(i+1,0)\}$ | $\{(i-1,1),(i,0),(i+1,0)\}$ |
| $\mathcal{L}(A_{12})$ | $\{(i,0)\}$ | $\{(i-1,1),(i,0)\}$ |

**Fig. 8.** Available Subscript Analysis Example

## 6 Load Elimination Transformation

In this section, we present the algorithm for redundant load elimination. There are two steps in the algorithm: *Register pressure moderation* described in Section 6.1, which determines a subset of the redundant loads for load elimination and *Code generation* described in Section 6.2, which eliminates the redundant loads from the loop.

The set of redundant loads in a loop is represented using *UseRepSet*, a set of ordered pairs of the form $(A_j[x], d)$, where the use $A_j[x]$ is redundant and $d$ is the iteration distance from the generator to the use. $d = 0$ implies an intra-iteration reuse and $d \geq 1$ implies an inter-iteration reuse. *UseRepSet* is derived from the lattice sets computed by available subscript analysis.

$$UseRepSet = \{\ (A_i[x], d) \mid \exists\ (y, d)\ \in\ \mathcal{L}(A_j),\ A_k = u\phi(A_i, A_j),\ \mathcal{DS}(x, y) = true\}$$

For the loop in Figure 3, $UseRepSet = \{(B_2[i], 1), (A_4[i-1], 1), (B_4[i-1], 1), (B_7[i], 0)\}$

### 6.1 Register Pressure Moderation

Eliminating all redundant loads in a loop may lead to generation of spill code which could counteract the savings from scalar replacement. To prevent this, we need to choose the most profitable loads which could be scalar replaced using the available machine registers. We define the most profitable loads as the ones which requires the least number of registers.

When estimating the total register requirements for scalar replacement, all redundant uses which are generated by the same reference need to be considered together. To do this *UseRepSet* is partitioned into $U_1, ...U_k$, such that generators

of all uses in a partition are definitely-same. A partition represents a set of uses which do not dominate each other and are generated by the same use/def. A partition $U_m$ is defined as follows, where *step* is the coefficient of the induction variable in the subscript expression.

$U_m = \{(A_i[x_i], d_i) \mid \forall\ (A_j[x_j], d_j) \in U_m, \mathcal{DS}(x_i + d_i \times step, x_j + d_j \times step) = true\}$

If the array index expression is loop-invariant, the number of registers required for its scalar replacement is one. In other cases, the number of registers required for eliminating all the loads in the partition $U_p$ is given by

$NumRegs(U_p) = \{d_i + 1 \mid (A_i[x_i], d_i) \in U_p\ \wedge\ \forall\ (A_j[x_j], d_j) \in U_p,\ d_i\ \geq\ d_j\}$

For the loop in Figure 3, the four elements in *UseRepSet* will fall into four different partitions: $\{(B_2[i], 1)\}, \{(A_4[i-1], 1)\}, \{(B_4[i-1], 1)\}, \{(B_7[i], 0)\}$. The total number of registers required for the scalar replacement is 7.

The partitions are then sorted in increasing order of the number of registers required. To select the redundant loads for scalar replacement, we use a greedy algorithm in which at each step the algorithm chooses the first available partition. The algorithm terminates when the first available partition does not fit into the remaining machine registers.

### 6.2 Code Generation

The inputs to the code generation algorithm are the intermediate representation of the loop body, the Array SSA form of the loop, and the subset of *UseRepSet* after register pressure moderation. The code transformation is performed on the original input program. The extended Array SSA form is used to search for the generator corresponding to a redundant use. The algorithm for the transformation is shown in Figure 9. A scalar temporary, $A\_t_x$ is created for every array access $A[i]$ that is scalar replaced where, $\mathcal{DS}(x, i) = true$. In the first stage of the algorithm all redundant loads are replaced with a reference to a scalar temporary as shown in lines 2-11 of Figure 9. For example the reads of array elements $B[i]$ in line 1, $A[i-1]$ in line 5, $B[i-1]$ in line 6 and $B[i]$ in line 11 of Figure 11(a) are replaced with reads of scalar temporaries as shown in Figure 11(b). The loop also computes the maximum iteration distance for all redundant uses to their generator. It also moves loop invariant array reads to loop preheader. The loop in lines 15-27 of Figure 9 generates copy statements between scalar temporaries and code to initialize scalar temporaries if it is a loop carried reuse. The code to initialize the scalar temporary is inserted in the loop preheader, the basic block that immediately dominates the loop header. Line 2-4 in Figure 11(b) is the code generated to initialize the scalar temporaries and lines 23-25 are the copy statements generated to carry values across iterations. The loop in lines 20-24 of Figure 9 guarantees that the scalar temporaries have the right values if the value is generated across multiple iterations. Lines 28-35 of Figure 9 identifies the generators and initializes the appropriate scalar temporaries. The generators are identified using the recursive search routine SEARCH, which takes two arguments: The first argument is a SSA function $A_j$ and the second argument is an index $i$. The function returns the set of all uses/defs which generates $A[i]$. The

**Input:** Input loop, Array SSA form of the loop and *UseRepSet*
**Output:** Loop after eliminating redundant loads

1: $maxd \leftarrow 0$
2: **for all** $(A_i[x], d)$ in *UseRepSet* **do**
3: Replace LHS := $A_i[x]$ by LHS := $A\_t_x$
4: **if** $d > maxd$ and $x$ is not a loop invariant **then**
5: $maxd \leftarrow d$
6: **end if**
7: **if** $x$ is loop invariant **then**
8: Insert initialization of $A\_t_x$ in the loop preheader
9: $UseRepSet \leftarrow UseRepSet - (A_i[x], d)$
10: **end if**
11: **end for**
12: **for all** $(A_i[x], d)$ in *UseRepSet* **do**
13: $n \leftarrow x$
14: $dist \leftarrow d$
15: **while** $dist \neq 0$ **do**
16: **if** $A\_t_n$ is not initialized **then**
17: Insert $A\_t_n := A\_t_{n+step}$ at the end of loop body
18: Insert initialization of $A\_t_n$ in the loop preheader
19: **end if**
20: **for all** defs $A_j[k]$ := RHS **do**
21: **if** $\mathcal{DS}(n, k)$ **then**
22: Replace the def by $A\_t_n$ := RHS; $A_j[k] := A\_t_n$
23: **end if**
24: **end for**
25: $dist \leftarrow dist - 1$
26: $n \leftarrow n + step$
27: **end while**
28: $genset \leftarrow$ SEARCH$(A_h, n)$ where $A_h$ is the $h\phi$
29: **for all** uses $A_j \in genset$ **do**
30: Replace the use by $A\_t_n := A_j[k]$; LHS := $A\_t_n$
31: **end for**
32: **for all** defs $A_j \in genset$ **do**
33: Replace the def by $A\_t_n$ := RHS; $A_j[k] := A\_t_n$
34: **end for**
35: **end for**
36: Introduce a $maxd$-trip count test for the scalar replaced loop

**Fig. 9.** Redundant Load Elimination Transformation Algorithm

```
1: procedure SEARCH(A, i)
2:     if A = hφ(A_1, .., A_k) then
3:         return ∪_{j=2,k} SEARCH(A_j, i)
4:     end if
5:     if A = φ(A_1, .., A_k) then
6:         return ∪_{j=1,k} SEARCH(A_j, i)
7:     end if
8:     if A = dφ(A_1, A_2) then
9:         if L(A_1) = {k} and DS(i, k) then
10:            return {A_1}
11:        else
12:            return SEARCH(A_2, i)
13:        end if
14:    end if
15:    if A = uφ(A_1, A_2) then
16:        if L(A_1) = {k} and DS(i, k) then
17:            return {A_1}
18:        else
19:            return SEARCH(A_2, i)
20:        end if
21:    end if
22: end procedure
```

**Fig. 10.** Subroutine to find the set of generators

| (a) Original Loop | (b) After Redundant Load Elimination |
|---|---|
| 1: **for** $i = 1$ to $n$ **do** | |
| 2: $t1 = B[i]$ | 1: **if** $n > 2$ **then** |
| 3: $t2 = A[t1]$ | 2: $A\_t_{i-1} = A[0]$ |
| 4: **if** $t2 > 0$ **then** | 3: $B\_t_i = B[1]$ |
| 5: $t_3 = A[i-1]$ | 4: $B\_t_{i-1} = B[0]$ |
| 6: $t_4 = B[i-1]$ | 5: **for** $i = 1$ to $n$ **do** |
| 7: $t_5 = t_3 + t_4$ | 6: $t1 = B\_t_i$ |
| 8: $A[i+1] = t_5$ | 7: $t2 = A[t1]$ |
| 9: **end if** | 8: **if** $t2 > 0$ **then** |
| 10: $t_6 = A[i]$ | 9: $t_3 = A\_t_{i-1}$ |
| 11: $t_7 = B[i]$ | 10: $t_4 = B\_t_{i-1}$ |
| 12: $t_8 = B[i+1]$ | 11: $t_5 = t_3 + t_4$ |
| 13: $t_9 = t_6 + t_7$ | 12: $A[i+1] = t_5$ |
| 14: $t_{10} = t_9 + t_8$ | 13: **end if** |
| 15: $A[i] = t_{10}$ | 14: $A\_t_i = A[i]$ |
| 16: **end for** | 15: $t_6 = A\_t_i$ |
| | 16: $t_7 = B\_t_{i-1}$ |
| | 17: $B\_t_{i+1} = B[i+1]$ |
| | 18: $t_8 = B\_t_{i+1}$ |
| | 19: $t_9 = t_6 + t_7$ |
| | 20: $t_{10} = t_9 + t_8$ |
| | 21: $A\_t_i = t_{10}$ |
| | 22: $A[i] = A\_t_i$ |
| | 23: $A\_t_{i-1} = A\_t_i$ |
| | 24: $B\_t_{i-1} = B\_t_i$ |
| | 25: $B\_t_i = B\_t_{i+1}$ |
| | 26: **end for** |
| | 27: **else** |
| | 28: original loop as shown in Figure 11(a) |
| | 29: **end if** |

**Fig. 11.** Redundant Load Elimination Example

SEARCH routine is given in Figure 10. The routine takes at most one backward traversal of the SSA graph to find the set of generators. Line 36 of the load elimination algorithm inserts a loop trip count test around the scalar replaced loop.

We now present a brief complexity analysis of the load elimination transformation described in Figure 9. Let $k$ be the total number of loads and stores of array elements inside the loop and let $l$ be the number of redundant loads. The algorithm makes $l$ traversals of the SSA graph and examines the stores inside the loop a maximum of $l \times d$, where $d$ is the maximum distance from the generator to the redundant use. Therefore the worst case complexity of the algorithm in Figure 9 for a given loop is $O((d+1) \times l \times k)$.

| **(a) Original Loop** | **(b) After Load Elimination** |
|---|---|
| 1: **for** $i = 1$ to $n$ **do**<br>2: $A[i+1] = e_1$<br>3: $A[i] = A[i] + e_2$<br>4: **end for** | 1: $A\_t\_init_i = A[1]$<br>2: **for** $j = 1$ to $n$ **do**<br>3: $A\_t_i = \phi(A\_t_{i+1}, A\_t\_init_i)$<br>4: $A\_t_{i+1} = e_1$<br>5: $A[i+1] = A\_t_{i+1}$<br>6: $A\_t_i = A\_t_i + e_2$<br>7: $A[i] = A\_t_i$<br>8: **end for** |
| **(c) Extended Array SSA** | **(d) After Store Elimination** |
| 1: $A_0 = ...$<br>2: $A\_t\_init_i = A_1[1]$<br>3: $A_2 = u\phi(A_1, A_0)$<br>4: **for** $j = 1$ to $n$ **do**<br>5: $A_3 = h\phi(A_2, A_7)$<br>6: $A\_t_i = \phi(A\_t_{i+1}, A\_t\_init_i)$<br>7: $A\_t_{i+1} = e_1$<br>8: $A_4[i+1] = A\_t_{i+1}$<br>9: $A_5 = d\phi(A_4, A_3)$<br>10: $A\_t_i = A\_t_i + e_2$<br>11: $A_6[i] = A\_t_i$<br>12: $A_7 = d\phi(A_6, A_5)$<br>13: **end for** | 1: $A\_t\_init_i = A[1]$<br>2: **for** $j = 1$ to $n$ **do**<br>3: $A\_t_i = \phi(A\_t_{i+1}, A\_t\_init_i)$<br>4: $A\_t_{i+1} = e_1$<br>5: $A\_t_i = A\_t_i + e_2$<br>6: $A[i] = A\_t_i$<br>7: **end for**<br>8: $A\_t_i = A\_t_{i+1}$<br>9: $A[i+1] = e_1$<br>10: $A\_t_i = A\_t_i + e_2$<br>11: $A[i] = A\_t_i$ |

**Fig. 12.** Store Elimination Example

## 7 Dead Store Elimination

Elimination of loads can increase the number of dead stores inside the loop. For example, consider the loop in Figure 12(a). The store of $A[i+1]$ in line 2 is used by the load of $A[i]$ in line 3. Assuming $n > 0$, Figure 12(b) shows the same loop after scalar replacement and elimination of redundant loads. The store of $A[i+1]$

| SSA function | Lattice Operation |
|---|---|
| $s_i : A_r = u\phi(A_q, A_p)$ | $\mathcal{L}_u(A_p, s_i) = \mathcal{L}(A_r) - \{(v,d) \mid \exists\ (w,0)\ \in\ \mathcal{L}(A_p)\ s.t.\ \neg\mathcal{DD}(v,w)\}$ |
| $s_i : A_r = d\phi(A_q, A_p)$ | $\mathcal{L}_u(A_p, s_i) = \text{UPDATE}(\mathcal{L}(A_r), \mathcal{L}(A_q))$ |
| $s_i : A_r = \phi(A_q, A_p)$ | $\mathcal{L}_u(A_q, s_i) = \mathcal{L}(A_r)$<br>$\mathcal{L}_u(A_p, s_i) = \mathcal{L}(A_r)$ |
| $s_i : A_r = h\phi(A_q, A_p)$ | $\mathcal{L}_u(A_q, s_i) = \text{SHIFT}(\mathcal{L}(A_r))$<br>$\mathcal{L}_u(A_p, s_i) = \text{SHIFT}(\mathcal{L}(A_r))$ |

**Fig. 13.** Index Propagation for Dead Store Elimination

| | Iteration 1 | Iteration 2 |
|---|---|---|
| $\mathcal{L}(A_7)$ | $\perp$ | {(i+1,1),(i+2,1)} |
| $\mathcal{L}(A_5)$ | {(i,0)} | {(i,0),(i+1,1),(i+2,1)} |
| $\mathcal{L}(A_3)$ | {(i,0),(i+1,0)} | {(i,0),(i+1,0),(i+2,1)} |

**Fig. 14.** Dead Subscript Analysis

in line 5 for the first $n-1$ iterations is now redundant since it gets overwritten by the store to $A[i]$ at line 7 in the next iteration with no uses in between.

Dead store elimination is run as a post pass to redundant load elimination and it uses a backward flow analysis of array subscripts similar to very busy expression analysis. The analysis computes set $\mathcal{L}(A_i)$ for every SSA function in the program. Similar to available subscript analysis presented in Section 5, the lattice for dead subscript analysis, $\mathcal{L}(A)$ is a subset of $\mathcal{U}^A_{ind} \times \mathbb{Z}_{\geq 0}$. Note that there could be multiple uses of the same SSA name. For instance, the SSA name $A_3$ is an argument of the $u\phi$ function in line 12 and the $\phi$ function in line 19 in the loop given in Figure 3(b). A backward data flow analysis will have to keep track of lattice values for each of these values. To achieve this, we associate a lattice element with each of the uses of the SSA variable represented as $\mathcal{L}_u(A_i, s_j)$, where $s_j$ is a statement in the program which uses the SSA variable $A_i$.

During the backward flow analysis, index sets are propagated from left to right of $\phi$ functions. The lattice operations for the propagation of data flow information are shown in Figure 13. The computation of $\mathcal{L}(A_i)$ from all the augmented uses of $A_i$ is given using the following equation.

$$\mathcal{L}(A_i) = \bigcap_{s_j \text{ is a } \phi \text{ use of } A_i} \mathcal{L}(A_i, s_j)$$

The lattice values are initialized as follows:

$$\mathcal{L}(A_i) = \begin{cases} \{(x,0)\} & A_i \text{ is a definition of the form } A_i[x] \\ \{(x,0)\} & A_i \text{ is a use of the form } A_i[x] \\ \top & A_i \text{ is defined outside the loop} \\ \perp & A_i \text{ is a SSA function defined inside the loop} \end{cases}$$

The SHIFT and UPDATE operations are defined as follows, where $step_1$ is the coefficient of the induction variable in $i_1$, $step_2$ is the coefficient of the induction variable in $i_2$ and so on.

$$\text{SHIFT}\langle(i_1, d_1), (i_2, d_2), \ldots\rangle = \langle(i_1 + step_1, d_1 + 1), (i_2 + step_2, d_2 + 1), \ldots\rangle$$

$$\text{UPDATE}((i', d'), \mathcal{L}(A_p)) = \{(i_1, d_1) | (i_1, d_1) \in \mathcal{L}(A_p) \text{ and } \mathcal{DS}(i', i_1) = false\} \cup \{(i', d')\}$$

The result of the analysis is used to compute the set of dead stores:

$$DeadStores = \{\ (A_i[x], d) \mid \exists\ (y, d)\ \in\ \mathcal{L}(A_j) \text{ and } \mathcal{DS}(x, y) = true \text{ and } A_k = d\phi(A_i, A_j)\}$$

*i.e.,* a store, $A_i[x]$ is redundant with respect to subsequent defs if $(y, d) \in \mathcal{L}(A_j)$ and $\mathcal{DS}(x, y) = true$, where $A_k = d\phi(A_i, A_j)$ is the $d\phi$ function corresponding to the use $A_i[x]$. $d$ represents the number of iterations between the dead store and the killing store.

Figure 12(c) shows the extended Array SSA form of the program in Figure 12(b). Figure 14 illustrates dead subscript analysis on this loop. The set of dead stores for this loop is $DeadStores = \{(A_4[i + 1], 1)\}$.

Given the set $DeadStores = \{(S_1, d_1), ...(S_n, d_n)\}$, the algorithm for dead store elimination involves peeling the last $k$ iterations of the loop, where $k = \max_{i=1..n} d_i$. The dead stores could be eliminated from the original loop, but they must be retained in the last $k$ peeled iterations. The loop in Figure 12(b) after the elimination of dead stores is given in Figure 12(d).

Similar to available subscript analysis, the worst case complexity of dead subscript analysis for a given loop is $O(\tau \times k)$. The complexity of the transformation is $O(n)$, where $n$ is the size of the loop body.

## 8 Extension to Objects and While Loops

In the previous sections, we introduced new scalar replacement analysis and transformations based on extended Array SSA form that can be used to optimize array accesses within and across loop iterations in counted loops. Past work has shown that scalar replacement can also be performed more generally on object fields in the presence of arbitrary control flow [10]. However, though the past work in [10] used Array SSA form, it could not perform scalar replacement across multiple iterations of a loop. In this section, we briefly illustrate how our approach can also perform inter-iteration scalar replacement in programs with while-loops containing accesses to object fields.

Figure 15(a) shows a simple example of a while loop in which the read of object field `p.x` can be replaced by a scalar temporary carrying the value from the previous iteration. This code assumes that *FIRST* and *LAST* refer to the first node and last node in a linked list, and the result of scalar replacement is shown in Figure 15(b). A value of $\tau = 1$ suffices to propagate *temp* from the

previous iteration to the current iteration, provided a prologue is generated that is guarded by a zero-trip test as shown in Figure 15(b). It is worth noting that no shape analysis is necessary for the scalar replacement performed in Figure 15(b). If available, shape analysis [20] can be used as a pre-pass to further refine the $\mathcal{DS}$ and $\mathcal{DD}$ information for objects in while loops.

| (a) Original Loop | (b) After Scalar Replacement |
|---|---|
| 1: $p := FIRST$ | 1: $p := FIRST$ |
| 2: **while** $p \neq LAST$ **do** | 2: **if** $p \neq LAST$ **then** |
| 3: ... = $p.x$; | 3: $temp = p.x$; |
| 4: ... | 4: **end if** |
| 5: $p = p.next$; | 5: **while** $p \neq LAST$ **do** |
| 6: $p.x$ = ... | 6: ... = $temp$; |
| 7: **end while** | 7: ... |
| | 8: $p = p.next$; |
| | 9: $temp = ...$ |
| | 10: $p.x$ = $temp$; |
| | 11: **end while** |

**Fig. 15.** Scalar replacement example for object accesses in a while loop

## 9 Experimental Results

In this section, we describe the implementation of our Array SSA based scalar replacement framework followed by an experimental evaluation of our scalar replacement and dead store analysis algorithms.

### 9.1 Implementation

We have implemented our algorithms in LLVM compiler release 3.2. A high-level view of the implementation is presented in Figure 16. To perform subscript analysis, we employed scalar evolution [17] as a pre-pass that computes closed form expressions for all scalar integer variables in a given program. This is followed by extended Array SSA construction, available subscript analysis, and redundant load elimination. Since there are $u\phi$s associated with the loads that were eliminated, an Array SSA repair pass is required after load elimination to cleanup the $u\phi$s and fix the arguments of control $\phi$s. The dead subscript analysis and dead store elimination follows the Array SSA repair pass. Finally, the program is translated out of Array SSA form.

### 9.2 Evaluation

Stencil computations offer opportunities for inter-iteration scalar replacement. We evaluated our scalar replacement transformation on 7 stencil applications: Jacobi 1-D 3-point, Jacobi 2-D 5-point, Jacobi 3-D 7-point, Jacobi 3-D 13-point,

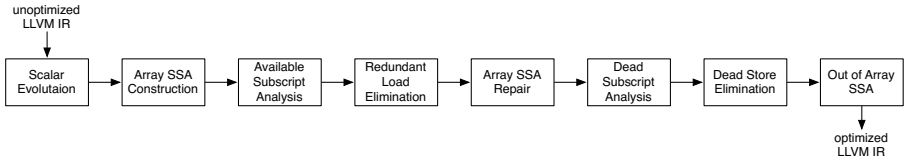


**Fig. 16.** High Level View of LLVM Implementation

Jacobi 3-D 19-point, Jacobi 3-D 27-point and Rician Denoising. For Jacobi 2-D 5-point example, we employed unroll-and-jam as a pre-pass transformation with an unroll factor of 4 to increase scalar replacement opportunities. No unrolling was performed on the remaining 3-D kernels, since they already contain sufficient opportunities for scalar replacement. We used $\tau = 5$, which is sufficient to capture all the load elimination opportunities in the applications.

The experimental results were obtained on a 32-core 3.55 GHz IBM Power7 system with 256 GB main memory and running SUSE Linux. The focus of our measurements was on obtaining dynamic counts of load operations[1] and the runtime improvement due to scalar replacement algorithms. When we report timing information, we report the best wall-clock time from five runs. We used the PAPI [15] interface to find the dynamic counts of load instructions executed for each of the programs. We compiled the programs with two different set of options described below.

- O3 : LLVM -O3 with basic alias analysis.
- O3SR : LLVM -O3 with basic alias analysis and scalar replacement

**Table 1.** Comparison of Load Instructions Executed and Runtimes

| Benchmark | O3 Loads | O3SR Loads | O3 Time (secs) | O3SR Time (secs) |
|---|---|---|---|---|
| Jacobi 1-D 3-Point | 5.58E+8 | 4.59E+8 | .25 | .25 |
| Jacobi 2-D 5-Point | 4.35E+8 | 4.15E+8 | .43 | .32 |
| Jacobi 3-D 7-Point | 1.41E+9 | 1.29E+9 | 1.66 | .74 |
| Jacobi 3-D 13-Point | 1.89E+9 | 1.77E+9 | 2.73 | 1.32 |
| Jacobi 3-D 19-Point | 2.39E+9 | 1.78E+9 | 3.95 | 1.72 |
| Jacobi 3-D 27-Point | 2.88E+9 | 1.79E+9 | 5.45 | 3.16 |
| Rician Denoising | 2.71E+9 | 2.46E+9 | 4.17 | 3.53 |

Table 1 shows the dynamic counts of load instructions executed and the execution time for the programs without scalar replacement and with scalar replacement. All the programs show a reduction in the number of loads when scalar

[1] We only counted the load operations because these benchmarks do not offer opportunities for store elimination.

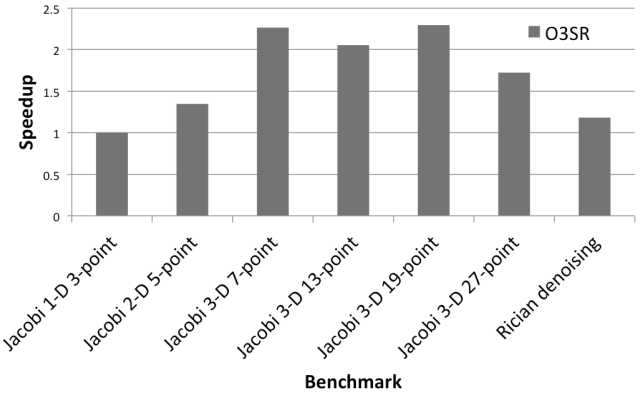


**Fig. 17.** Speedup : O3SR with respect to O3

replacement is enabled. Figure 17 shows the speedup for each of the benchmarks due to scalar replacement. All the programs, except Jacobi 1-D 3-Point displayed speedup due to scalar replacement. The speedup due to scalar replacement ranges from 1.18× to 2.29× for different benchmarks.

## 10 Related Work

Region Array SSA [19] is an extension of Array SSA form with explicit aggregated array region information for array accesses. Each array definition is summarized using a region representing the elements that it modifies across all surrounding loop nests. This region information then forms an integral part of normal $\phi$ operands. A region is represented using an uniform set of references (USR) representation. Additionally, the region is augmented with predicates to handle control flow. This representation is shown to be effective for constant propagation and array privatization, but the aggregated region representation is more complex than the subscript analysis presented in Section 5 and does not have enough maximum distance information to help guide scalar replacement to meet a certain register pressure. More importantly, since the region Array SSA representation explicitly does not capture use information, it would be hard to perform scalar replacement across iterations for array loads without any intervening array store.

A large body of past work has focused on scalar replacement [11,6,7,3,14] in the context of optimizing array references in scientific programs for better register reuse. These algorithms are primarily based on complex data dependence analysis and for loops with restricted or no control flow (e.g., [7] only handles loops with forward conditional control flow). Conditional control flow is often ignored when testing for data dependencies in parallelizing compilers. Moreover, [7] won't be able to promote values if dependence distances are not consistent. More recent algorithms such as [3,14] use analyses based on partial redundancy

elimination along with dependence analysis to perform load reuse analysis. Bodik et al. [4] used PRE along with global value-numbering and symbolic information to capture memory load equivalences.

For strongly typed programming languages, Fink, Knobe and Sarkar [10] presented a unified framework to analyze memory load operations for both array-element and object-field references. Their algorithm detects fully redundant memory operations using an extended Array SSA form representation for array-element memory operations and global value numbering technique to disambiguate the similarity of object references. Praun et al. [18] presented a PRE based inter-procedural load elimination algorithm that takes into account Java's concurrency features and exceptions. All of these approaches do not perform inter-iteration scalar replacement.

[5] employed runtime checking that ensures a value is available for strided memory accesses using arrays and pointers. Their approach is applicable across loop iterations, and also motivated the specialized hardware features such as rotating registers, valid bits, and predicated registers in modern processors.

[21] extend the original scalar replacement algorithm of [7] to outer loops and show better precision. Extensions for multiple induction variables for scalar replacement are proposed in [2].

[9] presents a data flow analysis framework for array references which propagates iteration distance (aka dependence distance) across loop iterations. That is, instances of subscripted references are propagated throughout the loop from points where they are generated until points are encountered that kill the instances. This information is then applied to optimizations such as redundant load elimination. Compared to their work, our available subscript analysis operates on SSA form representation and propagates indices instead of just distances.

## 11 Conclusions

In this paper, we introduced novel simple and efficient analysis algorithms for scalar replacement and dead store elimination that are built on Array SSA form, an extension to scalar SSA form that captures control and data flow properties at the level of array or pointer accesses. A core contribution of our algorithm is a subscript analysis that propagates array indices across loop iterations. Compared to past work, this algorithm can handle control flow within and across loop iterations and degrades gracefully in the presence of unanalyzable subscripts. We also introduced code transformations that can use the output of our analysis algorithms to perform the necessary scalar replacement transformations (including the insertion of loop prologues and epilogues for loop-carried reuse). Our experimental results show performance improvements of up to 2.29× relative to code generated by LLVM at -O3 level. These results promise to make our analysis algorithms a desirable starting point for scalar replacement implementations in modern SSA-based compiler infrastructures such as LLVM, compared to the more complex algorithms in past work based on non-SSA program representations.

## References


1. Polybench: Polyhedral benchmark suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/
2. Baradaran, N., Diniz, P.C., Park, J.: Extending the applicability of scalar replacement to multiple induction variables. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 455–469. Springer, Heidelberg (2005)
3. Bodik, R., Gupta, R.: Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures. In: Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 1–15. Springer, Heidelberg (1996)
4. Bodík, R., Gupta, R., Soffa, M.L.: Load-reuse analysis: Design and evaluation. SIGPLAN Not. 34(5), 64–76 (1999)
5. Budiu, M., Goldstein, S.C.: Inter-iteration scalar replacement in the presence of conditional control flow. In: 3rd Workshop on Optimizations for DSO and Embedded Systems, San Jose, CA (March 2005)
6. Callahan, D., Carr, S., Kennedy, K.: Improving Register Allocation for Subscripted Variables. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, White Plains, New York, pp. 53–65 (June 1990)
7. Carr, S., Kennedy, K.: Scalar Replacement in the Presence of Conditional Control Flow. Software—Practice and Experience (1), 51–77 (1994)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
9. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical data flow framework for array reference analysis and its use in optimizations. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI 1993, pp. 68–77. ACM, New York (1993)
10. Fink, S.J., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: Proceedings of the 7th International Symposium on Static Analysis, SAS 2000, pp. 155–174. Springer, London (2000)
11. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco (2002)
12. Knobe, K., Sarkar, V.: Array SSA form and its use in Parallelization. In: 25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages (January 1998)
13. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California (March 2004)
14. Lo, R., Chow, F., Kennedy, R., Liu, S.-M., Tu, P.: Register promotion by sparse partial redundancy elimination of loads and stores. SIGPLAN Not. 33(5), 26–37 (1998)
15. Mucci, P.J., Browne, S., Deane, C., Ho, G.: Papi: A portable interface to hardware performance counters. In: Proceedings of the Department of Defense HPCMP Users Group Conference, pp. 7–10 (1999)
16. Paek, Y., Hoeflinger, J., Padua, D.: Efficient and precise array access analysis. ACM Trans. Program. Lang. Syst. 24(1), 65–109 (2002)

17. Pop, S., Cohen, A., Silber, G.-A.: Induction variable analysis with delayed abstractions. In: Conte, T., Navarro, N., Hwu, W.-m.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 218–232. Springer, Heidelberg (2005)
18. Von Praun, C., Schneider, F., Gross, T.R.: Load Elimination in the Presence of Side Effects, Concurrency and Precise Exceptions. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 390–405. Springer, Heidelberg (2004)
19. Rus, S., He, G., Alias, C., Rauchwerger, L.: Region array ssa. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, PACT 2006, pp. 43–52. ACM, New York (2006)
20. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
21. So, B., Hall, M.: Increasing the applicability of scalar replacement. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 185–201. Springer, Heidelberg (2004)